Single System Image in a Linux-based Replicated Operating System Kernel

Akshay Giridhar Ravichandran

Dissertation submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Dr. Binoy Ravindran, Chair
Dr. Broadwater
Dr. Antonio Barbalace

February 24, 2015
Blacksburg, Virginia

Keywords: Linux, Multikernel, Thread synchronization, Signals, Process Management
Copyright 2015, Akshay Ravichandran
Recent trends in the computer market suggest that emerging computing platforms will be increasingly parallel and heterogeneous, in order to satisfy the user demand for improved performance and superior energy savings. Heterogeneity is a promising technology to keep growing the number of cores per chip without breaking the power wall. However, existing system software is able to cope with homogeneous architectures, but it was not designed to run on heterogeneous architectures, therefore, new system software designs are necessary.

One innovative design is the multikernel OS deployed by the Barrelfish operating system (OS) which partitions hardware resources to independent kernel instances that communicate exclusively by message passing, without exploiting the shared memory available amongst different CPUs in a multicore platform. Popcorn Linux implements an extension of the multikernel OS design, called replicated-kernel OS, with the goal of providing a Linux-based single system image environment on top of multiple kernels, which can eventually run on different ISA processors. A replicated-kernel OS replicates the state of various OS sub-systems amongst kernels that cooperate using message passing to distribute or access various services uniquely available on each kernel.

In this thesis, we present mechanisms to distribute signals, namespaces, inter-thread synchronizations and socket state replication. These features are built on top of the existing messaging layer, process or thread migration and address space consistency protocol to provide the application with an illusion of single system image and developers with the SMP programming environment they are most familiar with.

The mechanisms developed were unit tested with micro benchmarks to validate their correctness and to measure the gained speed up or additive overhead. Real-world applications were also used to benchmark the developed mechanisms on homogeneous and on heterogeneous architectures. It is found that the contributed Popcorn synchronization mechanism exhibits overhead compared to vanilla Linux on multicore as Linux’s equivalent mechanisms is tightly coupled with underlying hardware cache coherency protocol, therefore, faster than software message passing. On heterogeneous platforms, the developed mechanisms allow to transparently map each portion of the application to the processor in the platform on which the execution is faster. Optimizations are recommended to further improve the performance of the proposed synchronization mechanism. However, optimizations may force the user-space application and libraries to be rewritten in order to decouple their synchronization mechanisms of shared memory, therefore losing transparency, which is one of the primary goals of this work.
Dedication

This work is dedicated to my parents.
Acknowledgments

This work is supported in part by ONR under grant N00014-13-1-0317. Any opinions, findings, and conclusions expressed in this thesis are those of the author and do not necessarily reflect the views of ONR.
# Contents

1 Introduction 1
   1.1 Scalability Problem on Homogeneous ISA Platforms . . . . . . . . . . . . . . 2
   1.2 Scalability Problem on Heterogeneous ISA Platforms . . . . . . . . . . . . . 3
   1.3 Research Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
   1.4 Thesis Scope . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
   1.5 Thesis Organization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2 Related Work 5
   2.1 Single System Image on Multikernels . . . . . . . . . . . . . . . . . . . . . . 5
       2.1.1 K2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
       2.1.2 Cerberus . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
       2.1.3 Barrelfish Operating System . . . . . . . . . . . . . . . . . . . . . . 6
       2.1.4 Factored Operating System . . . . . . . . . . . . . . . . . . . . . . 7
   2.2 Single System Image on Multicellular Operating Systems . . . . . . . . . . . . 8
       2.2.1 Cellular IRIX . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
       2.2.2 Disco . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
       2.2.3 Hive . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
   2.3 Single System Image on Clusters . . . . . . . . . . . . . . . . . . . . . . . . 10
       2.3.1 Kerrighed . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
       2.3.2 MOSIX . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
       2.3.3 OpenSSI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
3  Popcorn Background
3.1  Popcorn System Overview
3.2  Multikernel Boot
3.3  CPU Partitioning
3.4  Memory Partitioning
3.5  Message Passing
3.6  Device Drivers
3.7  Thread Migration
3.8  Distributed File System

4  Signals
4.1  Introduction
   4.1.1  Signals During Start of a Thread
   4.1.2  Signals Delivery
4.2  Design Motivation
4.3  Implementation
   4.3.1  Signals Migration
   4.3.2  Signals Delivery
4.4  Evaluation

5  Futex
5.1  Introduction
   5.1.1  Futex Wait
   5.1.2  Futex Wake
   5.1.3  Futex Requeue
5.2  Design Motivation
5.3  Implementation
# List of Figures

3.1 Popcorn System Overview .............................................. 13

4.1 Distributed Signals control flow ....................................... 20

5.1 Kernel Space design of Futex in Linux ................................. 23
5.2 Data structure organization on the server side ....................... 27
5.3 Distributed Wait Control Flow ........................................ 29
5.4 Distributed Wake Control Flow ....................................... 30
5.5 Distributed Requeue Control Flow ..................................... 31
5.6 Spinlock Vs Linux Vs Popcorn ......................................... 36
5.7 Critical Section Access ................................................ 38
5.8 Relative comparison of costs in heterogeneous and homogeneous ... 40
5.9 Performance comparison for IS-BOMP .................................. 42
5.10 Performance comparison for CG-BOMP ................................. 43
5.11 Performance comparison for FT-BOMP ................................ 44
List of Tables

4.1 Test cases executed for evaluating signals ......................... 21
4.2 Cost of signals sent by kill ........................................... 21
4.3 Cost of signals sent by killall ........................................ 21

5.1 Futex's user space variable state .................................... 23
5.2 Test cases executed for evaluating Futex ........................... 34
5.3 Lock Scalability .......................................................... 35
5.4 Perf results for scalability .............................................. 35
5.5 Barriers executed for 100 Rounds ..................................... 36
5.6 Producer-Consumer: Number of access into critical section .... 37
5.7 Average syscalls and retries for producer-consumer ................ 37
5.8 Perf results for mutex between two threads ......................... 39
5.9 Mutex between two threads serialized by barriers ................. 39
5.10 Mutex between two threads serialized by barriers in Heterogeneous .. 39
5.11 Average syscalls and retries for Mutex between two threads .... 39
5.12 Syscall count comparison for IS-BOMP ............................ 41
5.13 Cost breakdown for IS-BOMP ......................................... 42
5.14 Syscall count comparison for CG-BOMP ............................ 43
5.15 Cost breakdown for CG-BOMP ......................................... 44
5.16 Cost breakdown for FT-BOMP ......................................... 45
5.17 Perf analysis of Fluidanimate ........................................... 45
5.18 Fluidanimate .............................................................. 46
5.19 Compression starts after entire file is read .......................... 46
5.20 Average syscalls and retries for PBZIP2 .............................. 47
5.21 Compression starts when the file is dynamically read ............... 47
5.22 Matrix multiplication ...................................................... 47

6.1 Average Response Time for ps command .............................. 52
Chapter 1

Introduction

To provide versatility in terms of performance, power, and chip area, heterogeneous ISA cores are increasingly consolidated into a single chip and share a single memory space. Traditional system software, that is mainly designed for symmetric multi-processing (SMP) hardware should be redesigned for these systems. In fact, CPUs with different ISA cannot run the same native compiled binary, neither a single monolithic kernel. Hence, traditional monolithic kernels cannot leverage the heterogeneous hardware. Processors with multiple-ISA cores are emerging in the mobile system-on-chip market. ARM big.LITTLE [1] is one of such emerging architectures that includes cache coherent memory access amongst heterogeneous CPUs. Big.LITTLE feature a high-performance Cortex-A15 processor and an energy efficient Cortex-A7 connected by a cache coherent interconnect, which enables any task to choose compute intensive or power efficient core based on their workload characteristic. Heterogeneity is also exploited in the desktop computer and server-class machines. Intel recently released a platform with a Xeon processor and a Xeon-Phi accelerator that interconnect over PCIe and feature overlapping ISA processors [2]. They use a separate stripped down operating system kernel per ISA to run applications.

These emerging platforms challenge system software developers to provide a unified system view to achieving scalability without forsaking the programmability and manageability. One step towards addressing scalability issues is the concept of multikernel. The multikernel OS design [3], introduced by Bauman et al., of the Systems Group at ETH Zurich and Microsoft Research, is implemented by the Barrelfish operating system. The multikernel design provides an alternative to the traditional SMP operating system kernel design based on shared memory. In a multikernel operating system, each kernel manages a private memory space, is assigned a subset of hardware peripherals, and runs on a processor core. Kernels do not share any memory or data structure, but they coordinate via message passing. This enables to keep the data structures of each kernel independent which is the key feature for a single environment to host different ISAs and access resources without any kernel synchronization. To exploit the available hardware or share resources, the OS state will be replicated and
kept coherent by message passing.

The Popcorn Linux operating system is an extension of the multikernel model geared towards building scalable operating systems for homogeneous and heterogeneous cores based on Linux. The available cores are partitioned across different individual Linux kernels based on ISA, NUMA node or simply one per kernel. These kernels can be assigned various hardware resource like physical RAM, file system, network interface and PCI devices. Each resource subset is isolated from the other. To share these isolated hardware entities, the kernels will cooperate in order to give an illusion of a single operating system. In order to support existing application and programming primitives as well as to map the correct hardware, the system should handle this cooperation in kernel space. In a distributed environment like Popcorn, there is a need for single system image as it offers the user for ease of programming, manageability, scalability and availability. As we can see, this setup inherently provides isolation and can be modified to cater availability. But the key feature for an operating system running on a diverse hardware to be robust are scalability and programmability. Scalability can be achieved by a thread or task migration and efficient messaging subsystem. Existing mechanisms for task and thread migration between kernels for effective workload balancing within the multikernel are described in section 1.1 and section 1.2.

1.1 Scalability Problem on Homogeneous ISA Platforms

A replicated-kernel OS reduces the contention on shared data structures by replicating the shared data on different kernels. By partitioning hardware resource management among different kernels each OS subsystem service is split into multiple servers. Each of them managing a slice of the OS subsystem, therefore reducing the contention on the single subsystem, improving system performance.

Popcorn Linux for homogeneous platforms has been shown to alleviate the bottleneck caused by the existing contention in the Linux memory management subsystem [4]. In Popcorn Linux compute and memory bound multi-threaded applications that continuously allocate memory, migrate their thread to different kernels. The partitioned memory handling shows to reduce contention in the memory management subsystem reducing application execution time up to 40%. To complete the migration an address space consistency protocol is provided. Though the inter-kernel communication can have substantial overhead due to the great quantity of messages flowing, benefits can be found in computing and memory intensive applications as shown in our previous work [5].
1.2 Scalability Problem on Heterogeneous ISA Platforms

The proliferation of accelerator processors, e.g., GPGPU, GPU, etc., enabled system developers to build heterogeneous platforms connecting the main CPUs with different ISA processors. Non-shared memory systems can be built by plugging PCIe based accelerators into x86 server machines. To exploit this emerging hardware in case of heterogeneous applications, thread migration has been supported in heterogeneous versions of Popcorn Linux [6]. This extends the homogeneous version of replicating address pages. This implements a new page coherency protocol, similar to MSI [7] cache-coherency protocol, that replicate the memory page content to keep the distributed process address space consistent. Such user space distributed shared memory provides sequential consistency. Considerable work is underway for compiler-based modifications so that a program can interact with the OS scheduler to decide the best thread placement on the platform.

While a distributed nature of the model allows applying algorithms from the distributed computing, as it suffers from similar problems like synchronization, distributed locking and naming services; introducing new system architecture specific programming interfaces. This thesis focuses on a subsystem which helps in achieving synchronization between the distributed threads without breaking the SMP programming paradigm. This will help achieve the ease of programmability in a distributed replicated system. Also a unified process space and distributed signal mechanism are provided to provide manageability of the distributed threads.

1.3 Research Contributions

This thesis contributes the following features to Popcorn Linux:

- A mechanism is designed and implemented for distributed signaling for both normal and migrated tasks across all the kernels.

- A mechanism is designed and implemented for supporting multi-threaded synchronization of distributed threads across kernels. The implementation provides an SMP-like distributed synchronization technique POSIX compliant.
• An analysis of the scalability and the overhead incurred by the inter-kernel synchronization layer is performed. Additionally, such layer is compared against SMP Linux both on homogeneous and on heterogeneous architectures.

• A mechanism is designed and implemented for process management features. This includes kernel discovery and architectural information, unique process identification space to provide the user space program the usual single operating environment. Additionally, single IP space is provided for sharing of the same NIC among the kernels and socket state migration of server threads.

1.4 Thesis Scope

This thesis compares and evaluates the single system image features added to the Popcorn Linux kernel with traditional SMP operating system. This thesis is strictly confined to features related to intra-process or thread groups. User space thread synchronization is provided only through POSIX (Portable Operating System Interface) constructs and not System V [8]. Implementing distributed shared memory mechanism in kernel memory space will be our future work. This is needed for implementing traditional single system image features like pipes, shared memory and message queue. The mechanism for distributed file system and file descriptor migration are research topics of other students.

1.5 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 provides information about related work. Attention is given towards single system image features that exists in cluster computing. Chapter 3 presents the concepts that are necessary for an understanding of Popcorn Linux. The details of existing Popcorn subsystems are listed. Chapter 4 details the design and implementation of distributed signals. Chapter 5 details the design and implementation of thread synchronization across kernels. Chapter 6 details the design and implementation process management features. Chapter 7 includes the conclusion and future work.
Chapter 2

Related Work

One of the main contributions of this work is an inter-thread synchronization primitive, based on distributed user space shared memory and user-kernel interaction, for user space tasks only, without breaking the POSIX API, thus enabling backward compatibility. This also provides process management features and signaling feature to stitch it into a single system image. In this chapter, we will examine existing techniques for single system image and we will study how these techniques have been applied in clustered environment as well as multikernels.

2.1 Single System Image on Multikernels

2.1.1 K2

This is a mobile operating system for heterogeneous coherence domains introduced by Rice University in 2014 [9]. In platforms with multiple coherency domains, two main approaches have been developed: shared-nothing with isolated many cores and shared-everything with coherent multicores. Instead, their philosophy is called share-mostly: a mid-of-the-way approach between the existent approaches to share necessary components. The kernels have both private and shared services.

They provide a single system image with 2 kernels running on OMAP4: the main kernel in ARM ISA running on Cortex-A9 cores, and the shadow kernel in Thumb-2 ISA running on a Cortex-M3 core. They provide global memory management, interrupt management, and global namespaces.

The synchronization is provided between process and not within thread groups. They argue that threads share various sets of OS state and running on multiple heterogeneous domains may lead to extensive contention. The strategy is to delay the light task execution thread
for a heavy task execution thread for sharing the resources. This is geared more towards the mobile environment and doesn’t give a generic framework for application developers to run different types of threading models. Also, this inhibits application which does a variety of tasks, each task being executed in multi-threaded mode, to exhibit multi-domain parallelism.

Interrupt management is done by directly hardwire interrupt pins to the respective kernels. This will not be the case in multiple instances of different heterogeneous kernels; which may be found in servers.

### 2.1.2 Cerberus

Cerberus is a parallel operating system introduced by Fudan University in 2011 [10]. Cerberus proposes scaling UNIX-like operating system by virtualization to provide backward compatibility for legacy applications, with the goal of reducing the contentions on the shared data structures by splitting them across virtual machines based on Xen. To share the resources like memory, file system and others, they use shared memory, state replication, message passing to coordinate access and reuse existing components. They have a single shared memory managed by a virtual memory monitor. For the operations that are shared between the virtual machine, they have incorporated system call virtualization. They intercept system calls from all virtual machines and forward them to the virtual machine monitor to resolve consistency problems. Since it is a virtualized environment, it allows a centralized layer to manage memory and consistency easily.

They support synchronization across processes in different virtual machines by virtualizing POSIX system calls. The calls made for remote addresses are converted to the real address to monitor and are stored in a common shared memory. Signals are supported by an underlying layer common to all virtual machines. Since the virtual machine monitor acts as a server, contention increases as the instances increase. Their target applications, including dbench and apache, are neither communication nor synchronization intensive thus showing some performance benefits under specific configuration. This configuration is not suitable for a heterogeneous platform where you cannot have a dedicated server.

### 2.1.3 Barrelfish Operating System

Barrelfish introduced the notion of multikernel in 2008. It is being jointly developed by the Systems Group at ETH Zurich, and Microsoft Research [3]. The main idea is to map an OS node to a CPU core. The goal is to make these nodes communicate asynchronously through message passing and replicate OS state to create unified userspace. They predicted this model would scale well for heterogeneous architectures as there will be shared data structures for various architectures. For dealing with heterogeneity, it employs a service known as the system knowledge base (SKB), which maintains all the information about the hardware
topology. They have hardware and device discovery by which information about PCI, device drivers, CPUID are collected and their properties are stored. They also get communication or interconnect topology during startup to construct topology-aware networks. Based on the architecture and interconnect available, it generates a suitable multicast tree for efficient communication between the cores.

Synchronization is implemented by the Octopus module which is a part of system knowledge base [11]. This provides locking primitives based on a key-value data structure called records stored in SKB. These are used for synchronization of a distributed application across kernels. Acquiring a lock creates a sequential record using the unique name agreed on by the clients. The client holding the least record number gets the lock. The other client broadcast a call to the previous record. When the lock is released by the holder, the next client in the queue will be triggered, therefore the client will be woken up.

Barriers are enforced using Octopus’s sequential records. Every client entering the barriers enters the sequential record and a check for the threshold number of records has reached. If so, it creates a new record communicating to all the clients that are ready to proceed.

2.1.4 Factored Operating System

The Factored Operating System (FOS) is designed by MIT Carbon Labs for multi-core platforms with thousands of cores [12]. Each core runs a microkernel hosting an application or system services. FOS distributes the services to different dedicated CPU cores and run applications in the remaining cores, thus isolation and fault tolerant is achieved for high availability. Services can be replicated to exploit the spatial locality of the cores for optimized application and service core interconnection.

They provide single system image services using Fleets [13], a scalable data structure. The name service, network stack, file system and distributed page allocation mechanisms use this for their services. Fleets are hosted in a single or few dedicated cores. There will be a fleet coordinator to which the application request services which then looks at the underlying service to map the request. This provides an abstraction of the underlying resource for the application to provide a single system image.

FOS proposed two methods to synchronize- lock servers and transaction servers. A synchronization lock can be managed by a server. In order to acquire the lock, messages are passed to the owner of the lock to provide access to critical sections. They realized this could be an overhead and limit performance. Thus, they provide transactional synchronization using transaction servers. This model will let the cores run the application in parallel and contact this server for transaction serialization to take advantage of optimistic concurrency. These synchronization mechanisms were introduced to keep access to kernel resources
consistent. This is a significant move away from SMP to get scalability with a large number of cores. However, the applications have to be rewritten to adapt to this architecture.

2.2 Single System Image on Multicellular Operating Systems

2.2.1 Cellular IRIX

Cellular IRIX was released by Silicon Graphics in 1998 [14]. Their main goal is scalability. This version utilizes multiple cells to scale up to thousands of CPUs. They offer single system image for users, administrators as well as running the application will have access to all the resources available within the connected cells. They achieve this by employing isolation and peer-to-peer mechanisms. This is implemented by the following architecture:

1. Golden cells will run some specific subsystem and no application is run on this. They have their networking subsystem in the golden cell which has access to the network interface. Thus, all network requests will be redirected.

2. Subsystems like file system, disk management are distributed by peer-to-peer mechanisms. They have a distributed file system called cellular XFS which manages the disk related issues across cells.

This system is intended to run parallel applications across cells. They provide shared memory transparently across cells; the application is given direct access to this memory. So without the intervention of operating system, they can share or transfer messages and data. Their target applications are shared memory based MPI programs. This will have better performance over clusters as they don’t have to share the state with message passing and can compare it with normal SMP. They have shown performance benefits by replacing the communication channels with shared memory. They support process synchronization which also utilizes the shared memory to reach consensus between two processes across different cells. They don’t support thread synchronization which are written with pthread and intra-thread synchronization is confined to individual cells. This architecture suffers from high contention by having a single spinlock to protect the kernel resources.

2.2.2 Disco

Disco [15] is a scalable virtualization-based operating system to host client virtual machines on unsupported hardware. Disco introduces a software layer which virtualizes the underlying hardware. Disco can host many independent client virtual machines and they are connected
through virtual Ethernet network to form a cluster. Disco virtualizes CPU, I/O peripherals and provides distributed shared memory.

For load balancing purposes, the virtualized CPU’s are shared among different client instances for efficient utilization of underlying hardware. As with the I/O devices, Disco intercepts the communication to and from the I/O devices. To support efficient communication, the monitor virtualizes access to the networking devices of the underlying system. Each virtual machine is assigned a distinct address with a virtual subnet. Disco supports ethernet devices without fragmentation for large data throughput. For one of the clients to access the outside world, it acts as a gateway using the physical network interfaces of the machine to send and receive packets.

This implementation is limited to Stanford’s FLASH SMP machines. As with any virtualized approach, it is easier to implement SMP/NUMA support with a dedicated server for all the services in the underlying layer than implementing at the OS level. But it is not suitable for a heterogeneous platform where the services has to be distributed to handle different underlying hardware.

2.2.3 Hive

Hive is an OS designed for SMP and is implemented on the Stanford’s FLASH multiprocessor [16]. It is comprised of many kernels communicating through a network. Each kernel is assigned or partitioned a specific number of cores, memory and I/O devices. Process management is distributed so users and applications see a single-system image. When a process forks a new process is created in a remote kernel and they interact with each other through RPC. For example, a kernel sending a signal to a process or process group may send messages to other cells in order to complete the operation. Single system image functionalities provided are limited. It includes System V streams (which includes interprocess pipes and network connections) and shared memory.

It supports multi-threading by a mechanism called spanning tasks. A spanning task is a process with threads running on multiple kernels. They keep the address space coherent to support task migration. Synchronization is provided between kernels. However, differently from Popcorn (and Linux) they do not support POSIX based multi-threading though they have an SMP hardware.
2.3 Single System Image on Clusters

2.3.1 Kerrighed

Kerrighed offers a single system image, like in an SMP machine, on top of a network of PCs using Linux container (lxc). It provides process migration, checkpointing, global scheduling, distributed file system and distributed IPC and streams. Global process and resource space are inherently obtained by associating with the container. It employs a centralized master node to store process states, shared resources like memory and devices. Kerrighed starts on a master node on which a Kerrighed container is started, the container loads an ssh daemon listening on a common port. The slave machines will also start the container and connect with the master.

The container gives an illusion of distributed shared memory (DSM). This is implemented at kernel space to achieve sequential consistency with a page granularity. All the above-mentioned features are provided by shared object space created by Kernel Distributed Data Management started in the master. This acts as the directory for any services. Every service is recorded as an object in this layer.

Support for POSIX synchronization primitives is directly provided by the Elrond module [17]. This module provides distributed locks, barriers, semaphores and wait conditions. This is done by libgthread which access the DSM to provide support for SMP locks. The functional aspect on how it is implemented and tests results are not available.

Socket or stream migration is supported in the context of process migration. A user space abstraction called KerNet [18] sockets is responsible for socket migration between or with the node. The application is associated with a KerNet socket; it then maintains the state and can perform actions like suspend and resume.

2.3.2 MOSIX

MOSIX [19] operating system was created by Amnon Barak and Amnon Shiloh from the Hebrew University at Jerusalem to provide single system image for clusters and clouds. Their main goal is to give an abstraction of configuration information and available resource of all the nodes and keep the user believe they are operating on a single system. They do automatic resource discovery by continuously monitoring the states and process migration and checkpointing for load balancing. They do provide distributed shared memory only for process migration through a patch [20]. They do not migrate threads.

MOSIX employs deputy mechanism for the migrated process with communication context. The main process in the original machine will redirect the packets or messages to the migrated
task. This will directly decrease the performance.

2.3.3 OpenSSI

OpenSSI is based on the Linux operating system and was developed by Compaq in 2001 [21] based on LOCUS [22] and is currently maintained by HP. Their contributes is to provide high availability, manageability, and scalability.

For load balancing, they provide process migration extending MOSIX. They support process migration, migrate IPCs like socket, pipes, shared memory and reopen these file in the remote. They have a distributed network file system using a cluster file system protocol. For manageability, they support distributed virtual file system for both process and device including cluster wide process identifier and signaling. For availability, they provide process checkpointing and restart.

2.3.4 Bproc

The goal of BProc [23] is to unify cluster of networked PC into a single machine using the master-slave architecture.

BProc provides process migration mechanisms. The master node manages the pool of distributed process identifiers and allocates every node portion of it. When a node wants to migrate a process, it informs the master node. It takes the process space of the slave node where the program is currently running and places it in the slave node where it wants to be migrated. After migration, when the new process is created in the slave node, it ignores the process ID that the slave node is assigning and attaches the original process identifier and all the PID system calls are changed to provide this secondary identifier. This also supports distributed signaling by means of ghost process redirecting the signal to the target node where the process is currently running.

2.3.5 Helios

The Helios Parallel Operating System [24] was written by members of the Helios group at Perihelion Software Limited released in 1988. The Helios Parallel Operating System has been built to run in parallel networked machines.

The design goals of Helios are to provide a unified operating system for parallel computers, independent of any specific hardware. They are built as a micro-kernel providing process management and migration using a distributed shared memory. The synchronization primitives provided are for inter-process communication and support migrated IPC.
This assigns to each node a specific purpose. Some nodes are nodes for computational purpose and some are I/O or Network nodes. Helios also isolates the underlying resource from the application to provide a single system image.
Chapter 3

Popcorn Background

We deployed Popcorn Linux on an homogeneous-ISA platform built with multicore AMD processors and on a heterogeneous-ISA platform built with Intel Xeon processors and an Intel Xeon Phi processor connected via PCIe.

3.1 Popcorn System Overview

Figure 3.1 shows the Popcorn system architecture and its sub-components. They will be briefly discussed in the following sections.

3.2 Multikernel Boot

In a homogeneous platform (e.g., a multicore processor), a single kernel is booted first with a single CPU or a cluster of CPUs. This acts as the primary kernel. The secondary kernel
instances can be booted by the user at any time in any given order. In our heterogeneous-ISA setup, the Xeon is booted first and then the Xeon-phi.

### 3.3 CPU Partitioning

In a homogeneous setup, CPUs are assigned to specific kernel instances at boot time. This is achieved via kernel boot params. The classical way of assigning CPUs to kernels would be to host a single kernel on a single CPU. A clustered configuration is possible by assigning more than a CPU to a single kernel and assign the first CPU in the list as a master to communicate with the other kernels. The cores can be assigned to kernels based on the platform topology, for example by loading a single kernel on all cores that share a single memory controller.

In the heterogeneous setup, CPUs belonging to a single ISA are grouped and assigned to a kernel. We have Xeon x86-64 cores in one kernel and Xeon-phi cores in another kernel.

### 3.4 Memory Partitioning

In a homogeneous setup, each kernel instance is configured with its own region of main memory at the boot time. Be it clustered or normal Popcorn, they are statically partitioned with the `memmap` Linux’s kernel boot option during load time. Currently, there is no global kernel space memory allocation available. The thread and process migration deals with the userspace distributed shared memory mechanism. So some physical memory will have to be shared when we migrate threads and are maintained by local data structures, in each kernel, pointing to the same physical memory. The coherence is guaranteed by the hardware. Some part of the main memory is also used for message passing between kernels.

On the heterogeneous platforms, each kernel has its own memory island. The pages will be replicated in the kernel where the thread is getting migrated and are kept consistent by a software layer which ensures sequential consistency. Although in the Xeon-Xeon Phi setup Xeon RAM can be mapped to the Xeon Phi’s physical address space and the contrary, this was not exploited because remote memory access is slower than local RAM access.

### 3.5 Message Passing

The kernel instances share the OS state by explicit message passing. Every software module, like task migration, address space migration, consistency algorithm, and the modules that are built for this thesis like synchronization, namespaces and signals utilizes this layer for all communication between kernels.
In homogeneous setups, we use a lock-free shared memory based message passing module to communicate between kernels. The messaging layer was designed and implemented by Ben Shelton [25]. This layer exposes a set of APIs to communicate between kernels. Every message is associated with a type and corresponding handler function. These must be registered during the kernel boot. When a kernel boots, it registers a receive queue in a shared memory location visible to all the kernels.

Sending the message is done by atomically claiming a ticket on the slot of the registered queue. A ticket is active when the assigned transfer slot is marked free for use. When the ticket is active, the sender places the data in the slot and marks the slot full and then generates an IPI to signal the receiver that a message is waiting for it.

Receiving message involves two steps. First, a CPU IPI handler function receives the message and executes a small code in bottom-half softirq. This notifies the top-half that messages are ready to be consumed. The top-half in work queue context executes the handler function registered for the message type. Once that function is executed, the message is consumed and is set free. Then the slot can be marked empty for further reuse.

The messaging API allows the developer to send short messages and long messages. The long messages are sent in the form of several short messages. The delivery is fast if we send short messages compared with long messages as the destination kernel has to wait for all the chunks to be there before delivering.

In the heterogeneous platform, we use Intel socket based interface to communicate between Xeon and Xeon-phi through PCIe interconnect.

### 3.6 Device Drivers

In a homogeneous setup, each device is controlled by a single kernel instance. For all the other kernel instances, these are blocked during the bootup time. The kernel instance that owns the device driver performs all driver operations. In heterogeneous, Xeon and Xeon-phi have individual access and control to its own set of devices. The shared access to devices is subjected of future research.

### 3.7 Thread Migration

Thread migration in Popcorn Linux for homogeneous platforms has been designed and implemented by David Katz [5]. When a thread is migrated between two kernels, a new thread
is created in the remote kernel, or an available thread is picked from a pool. The thread on
the origin kernel is paused, such thread is now called shadow. The process address space is
also copied over to the destination kernel, on multicore, we decided to only copy the data
structures that define the address space layout, and not the content. Address space migra-
tion can be on demand or up-front. Therefore, multiple kernels will have a page pointing to
the same physical address. The state is kept coherent by hardware cache coherency protocol.

In the heterogeneous setup, thread migration was designed and implemented by Marina
Sadini [26]. When the threads are migrated, a main thread group user thread will be cre-
ated. This will fork a new thread and then the new thread will get to user space. This main
thread will never go to user-space and acts as a manager thread for subsequent migration
requests. This is created as a user thread as it has to share the \texttt{mm_struct} to clone a child
thread to be a part of the same thread group. The address space layout (VMA), including
the stack and the heap, is also transferred similarly to the homogeneous case. The heteroge-
neous version differs from the homogeneous version due to the fact that in the latter memory
pages are kept coherent by the cache coherency protocol while this is not true in the former
setup.

### 3.8 Distributed File System

We have a distributed file system built on top of NFS. The mount namespace relies on NFS.
This module supports the file descriptor migration and consistency.
Chapter 4

Signals

4.1 Introduction

In Linux, signals are the main asynchronous communication mechanism between threads and processes, user-space and kernel-space. Signals are raised by I/O, software exceptions and generated for process and resource control.

4.1.1 Signals During Start of a Thread

Linux takes care of signals raised during the life cycle of a thread. If there is any pending signal during the process of forking, then the child thread creation will be stopped. The shared pending queue is shared within a thread group, but each thread has its own associated handler function and blocked queue. Once the thread is forked, the initial signal disposition will be set to ignore or to default for any thread. Also, the forked thread or the invoked process will lose the registered signal actions that the parent may have. If a task bearing a signal handling function for a signal forks a new child, this function has no meaning in the new process. The disposition of the signal is set to the default one in the new thread. If parent and child threads want to share a signal, the thread should be cloned with CLONE_SIGHAND, and then the signal handlers will be copied to its child.

4.1.2 Signals Delivery

Signals can be delivered synchronously or asynchronously. Synchronous signals occur during the threads execution flow. Asynchronous signals are caused due to external agents or interrupts coming from another process.

If a signal occurs, either from user space or kernel space, the signal is packed with its signal
information and user privileges as `struct sigqueue` and queued in the pending or shared queue of the thread. This can be done by `kill(pid, info)` if it is by user or `do_send_specific` and `force_sig` if it is from kernel space. When the kernel context switches it will check for the rescheduling flags. During this check, the kernel verifies if the thread has any pending signals. The function `do_signal` dequeues the signal from the queue and `handle_signal` is used for its disposition.

The thread has to configure the kernel with the appropriate action on how the signals can be handled:

1. **The signals can be blocked.** Signals will be added to blocked signal queue and not be added to pending signal handler queue until the application unblocks it. This is used to not deliver the signals.

2. **The signals can be ignored.** Signals will be added pending signal handler queue. But the default actions will be ignored.

3. **The signals can be caught.** The signal handler functions are registered by the thread with the kernel. This function is called by the kernel when that signal occurs. The thread can choose to terminate the program gracefully or do any other operation if it is not a fatal signal.

4. **The signals can be allowed to do default actions.** Every signal has a default action. This could result in thread getting terminated, or core dump or thread getting stopped.

Only `SIGKILL` and `SIGSTOP` can’t be ignored or blocked. This is because these two signals provide the mechanism to control/terminate the program under any circumstance.

Signals are delivered only to a single thread in any process. Apart from the hardware exceptions or the timer expiry (which are delivered to the thread which caused the event) all the signals are passed to one arbitrary thread of the process. Every thread has its own private signal mask which is used to block certain signals. So each thread can define how the signal can be handled.

Care should be taken in writing a handler function. This should be a re-entrant function as successive signals can schedule the thread again and again, preempting the function execution without entirely finishing the entire function. In practice, it should not hold static data over successive calls so that it should not return a pointer to static data, and uses the data provided by the caller or the local data. If global data should be accessed, we need to ensure protection. Also, it should not call any non-reentrant functions.

### 4.2 Design Motivation

The signals delivery and functions involving signals should be able to perform in a distributed environment as Popcorn. The following are the three necessary cases to accomplish the
implementation of distributed signals.

1. Signals should be redirected to either remote processes or migrated processes, generated from user or kernel space.
2. Migrated threads should have the same signal masks as the shadow threads.
3. Signals handling in Popcorn should be handled as in vanilla Linux, also during cloning of a process.

### 4.3 Implementation

Distributed signals in Popcorn are provided through mailboxes. Each kernel has a kernel thread which acts as an end point to redirect messages to the intended process and utilizes the existing messaging layer for communications. This thesis contributes signal state migration and signal delivery for normal and migrated tasks in remote kernels.

#### 4.3.1 Signals Migration

Along with the virtual memort aread (VMA) descriptors of stack and heap, the signal parameters are also copied to facilitate signal delivery in the case of migrated threads. The following parameters are copied across the kernel when the thread is migrated.

1. The blocked queues which contain already blocked signals.
2. The saved signal mask queue which contains all the signals that are blocked.
3. The signal pending and shared pending queue.
4. The stack pointer of the handler function.
5. The stack size of the handler function.
6. The registered signal actions for all the available signals.

#### 4.3.2 Signals Delivery

Figure 4.1 explains the control path of signals delivery under different scenarios. Since we have the process identification space partitioned, identifying the kernel is done by unmasking the higher order bits from the process identifier. struct siginfo is modified to accommodate remote flags. This field is required to override the user privileges in the remote kernel
once the signal is placed in the pending queue of the relevant process. When the thread is getting migrated, there will be a notification message updating the process identifier in the remote kernel to the shadow process. Whenever the migrated thread cascades to other kernels, it will notify both the shadow and the origin kernel. The origin kernel’s mailbox will act as the server for redirecting it to the appropriate kernel where the thread is alive. In case of heterogeneous replicated kernels, we create a shadow main user space thread which is used only to delegate work to the actual user threads and never actually go to user space. All the signals, including signals involving group exit, will be blocked reaching this thread as this is a management thread. During the normal group exit, these threads are reaped by the thread exit protocol. The real time signals delivery or timed signal deliveries are left as future work.

![Flowchart of Distributed Signals control flow](image)

**Figure 4.1: Distributed Signals control flow**

### 4.4 Evaluation

Popcorn’s distributed signals exploit Popcorn’s messaging layer for their delivery. The cost of messaging layer is well documented in our previous work [25]. To test the functionality of distributed signals we used `kill()`, `raise()`, `alarm()` API, targeting processes migrated or created on a remote kernel. Test cases mentioned in Table 4.1 from glibc test suite [27] (release 2.13) were used to validate the signal functions in the case of migrated threads.

The perceived latency from the user for issuing the `kill()`, `killall()` depends on the underlying messaging layer. These signals are sent synchronously with the signal info sent
Test Cases | Description
--- | ---
tst-signal5.c | Tests for `pthread_sigmask` for a migrated thread.
tst-signal6.c | Tests for use of alternate stack by handler function for migrate thread

Table 4.1: Test cases executed for evaluating signals

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time(seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>0.001</td>
</tr>
<tr>
<td>Remote</td>
<td>0.007</td>
</tr>
</tbody>
</table>

Table 4.2: Cost of signals sent by `kill`

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time(seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>0.033</td>
</tr>
<tr>
<td>Remote</td>
<td>0.051</td>
</tr>
</tbody>
</table>

Table 4.3: Cost of signals sent by `killall`

through messaging layer and placed in the queue of the corresponding process. If there is any error placing it, it will return the message to the local kernel with the error message. This is required for the user to have the same result as they will get for a local process. The average user time taken for a response with homogeneous messaging layer for signals delivery within local kernel and to remote kernels are mentioned in the Table 4.2 and Table 4.3 for `kill()` and `killall()` signals respectively. These are the average time measured by delivering signals to 1 through 32 kernels for 100 iterations.
Chapter 5

Futex

5.1 Introduction

In UNIX systems, System V IPC (inter-process communication) such as semaphores, msg-queues, sockets are the basic mechanisms for two processes to communicate. System V IPCs require the user program to issue a system call letting the kernel taking care of the communication irrespective of the degree of contention. Many multi-threaded applications use POSIX for synchronization between threads. Many common synchronization mechanisms are provided by Pthread, maintained by glibc. Amongst them, the Futex-based lock will not resort to system calls except when the lock is contended; since arbitration between tasks is low in real world applications, these mechanism provides low-latency and high-throughput.

Pthread provides APIs for:

1. Mutex locks
2. Condition variables
3. Read-write locks

The above API’s are constructed on top of FUTEX (Fast Userspace Mutex) module in kernel space in Linux [28]. Futex were introduced for gaining performance benefits on SMP machines and an alternative to semaphores and spinlocks. When using spin locks for inter-thread locking in user space, a thread is making the CPU busy without making any progress. On the other hand when we inform the kernel for the thread to sleep, it releases the CPU cycles to other processes improving responsiveness. Futex uses a unique integer variable (32 or 64 bit), which is in control of userspace, for synchronization. The reference count on the mm_struct is incremented, so it will exist as long the process is alive. All the above locking
Table 5.1: Futex’s user space variable state

<table>
<thead>
<tr>
<th>Value</th>
<th>State</th>
<th>Syscall</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Unlocked</td>
<td>No Syscall</td>
</tr>
<tr>
<td>1</td>
<td>Locked, Un-contended</td>
<td>No Syscall</td>
</tr>
<tr>
<td>2</td>
<td>Locked, Contended</td>
<td>Syscall</td>
</tr>
</tbody>
</table>

primitives are implemented by the following syscall.

long sys_futex(void *addr1, int op, int val1,
struct timespec *timeout, void *addr2, int val3);

Futex operates on the value of the address in the userspace according to Table 5.1 [28]:

When the syscall is made, it queues or dequeues the request in a hash bucket called Futex_hash_bucket as seen in the Figure 5.1 Futex_Updates.

There are three common kernel operations: Futex wait, Futex wake and Futex requeue called by glibc library for implementing their synchronization primitives.
5.1.1 Futex Wait

This operation enables the thread to be suspended in kernel space. We here describe the steps involved. First, the task gets the hash bucket pointer corresponding to the user space address and acquire the lock. Second, the tasks read the address content and checks the value. If it is modified then it would return -EWOULDBLOCK to user space. Else it would queue the thread and schedule it to sleep. When the wake signal is issued, it is dequeued and goes to user space returning success. If there is any other interrupt or signal, it would send -ERESTARTSYS and it returns to user space.

5.1.2 Futex Wake

This operation is used to wake up one or more threads queued against the same user space address in the hash bucket. We here describe the steps involved. First, the task gets the hash bucket pointer corresponding to the user space address and acquire the lock. Then it queries the queue and wakes up the sleeping task. It repeats the step until the required number of tasks, val1 (as described in Section 5.1), in the queue has been woken up. In the end, it releases the lock and returns the number of threads woken up to user space.

5.1.3 Futex Requeue

This is an optimized version of the wake operation used to mitigate thundering herd problem. When there are many threads waiting on a single lock, waking up everybody to check on their address value will result in most of them being queued back again. So this operation is used to wake a certain amount of threads queued in addr1 and the remaining will be queued to the addr2 which is also passed as an argument as described in Section 5.1. We here describe the steps involved. First, the task gets the hash bucket pointer corresponding to the both the user space addresses and acquire a lock. Then it queries the queue corresponding to address 1 and wakes up the sleeping task. Then it repeats the step until the required number of tasks, val1 (as described in Section 5.1), in the queue has been woken up. Then it transfers the remaining number of threads queued in addr1 to addr2. In the end, it releases the lock and returns the number of threads woken up to user space.

5.2 Design Motivation

Thread migration across different kernels breaks the existing synchronization constructs on monolithic kernel and calls for a distributed algorithm to tackle the problem.
Synchronization in non-shared memory systems is a well-researched area. One such method is by using a centralized lock server. It is easier to implement but cannot be used for architecture like Popcorn. The other method of synchronization is by using message passing without a server. It involves state replication and broadcast messages to all the nodes or nodes are connected in structures for convenient message passing, use logical timestamps to resolve contention when there is no global clock available. The algorithms are similar to Lamports mutual exclusion and they will consume messages in the order of $x \times (N - 1)$, where $x$ can vary between 1 to $N$, for resolving single contention. To maintain compatibility with existing applications and maintain the distributed nature to cater to non-shared memory systems, we decided to use distributed shared memory for emulating SMP synchronization.

We decided to support Pthread by modifying Futex kernel module with the help of the following available infrastructure.

1. We have user space address consistency to make use of the fast path for non contended cases.

2. Almost all applications are based on glibc, which will lend the easy programming aspect for the end user.

3. The normal Futex operation uses FIFO ordering and we have fast messaging layer which supports FIFO ordering.

### 5.3 Implementation

SMP Linux handles concurrent requests by means of holding the record of sleeping threads in a table of queues (Linux’s Futex queues) and this is protected by a kernel space spin-lock. The serialization is ensured by this mechanism for a monolithic kernel. Since we have threads distributed in multiple kernels, we need to preserve the serialization for every unique identifier the threads may use. The following design decisions are made to Futex for working in a distributed environment.

During the bootup of the kernel, the memory is statically partitioned. When the kernel is booted, the `memmap` argument is given as boot argument for assigning the range of memory. By parsing this, we can get the start of the memory address (`kernel_start_addr`) allocated for the kernel. Whenever a new kernel is booted up it broadcasts its range of page frame number (PFN) by reading the above and gets all the other kernel page frame number ranges. Every kernel does the same when it is booted up sharing a global list of memory ranges that each kernel holds.
The kernel where the userspace address is initialized will have an associated `struct page` to it. This kernel becomes the server for all the wait and wake operations that are related to that address. Every distinct memory regions of the process are managed by a kernel data structure called virtual memory area (VMA). So this userspace address can be found as a part of memory region held by the process. The local and migrated threads can determine the server kernel by checking the VMA flags and doing the page walk [29] to find the page frame number for the user space address.

For the client threads in a server kernel, the page frame number will be in the valid range of the current kernel with the page being a normal page or a copy on write (COW) page. For client threads in remote kernel the VMA flags for the migrated stack or heap will be of type VM_PFNMAP—VM_MIXEDMAP denoting the fact that it will have special pages only and they don’t hold the original page for the address. It defaults to check the global list to locate the server kernel when the page frame number is not valid or not in the range of current kernel.

When the memory regions are from different physical memory spaces, as in the heterogeneous system, this information is saved in the replicated page. An owner field is added to `struct page`. When the first thread access the user address, page walkthrough is done to acquire the latest copy of that page. Then the owner field is set to the kernel. Now this acts as the server kernel. When the other thread wants to lock on the user address, they access the latest page and know if it has an owner kernel or not. If so, the distributed Futex is kick started.

On the client side, every address on which the thread calls Futex syscall will have its own local state maintained by the following data structure. Based on the combination of thread group and user space address, a local request queue will be allocated. Apart from `pid` and `uaddr`, it will hold the `request_id` which will be atomically incremented. The `wait_queue` is used to put the current thread in sleep until the server responds with an event. The `operation` refers to either wait or wake requests. Once the server return `errnum` is updated with the return value. The `wake_state` refers to the wait request becoming invalid as it has already woken up. Each client requests can be in 4 states. Initially when the Futex syscall is made its state is set to IDLE. Then, when it waits for the ticket from the server, its status will be set to IDLE to INPROG. When the client receives the response from the server the status will be set to DONE. Finally, before invoking a call to `schedule` for sleeping, it changes its state to SLEEP.

```c
struct local_request_queue {
    pid;
    uaddr;
    request_id;
    wake_state;
    wait_queue;
    enum {
```
IDLE, INPROG, DONE, SLEEP

The server side has one global request handler thread assigned per thread group ID and per user space address and is maintained by a hash table data structure 5.2.

Figure 5.2: Data structure organization on the server side

The key for distributed synchronization is achieved by global ticketing mechanism for ordering the request, and granting the possibility to enter the critical section.

5.3.1 Global Ticketing Mechanism

In order for the client threads from all the kernels to wait or wake, they issue a call to the server, through the function `global_spinlock`, for acquiring the global ticket (note that the function doesn’t implement a spinlock). This utilizes Popcorn messaging layer which enforces first in first out ordering. The reason to couple the implementation with the FIFO messaging layer is that it is easier- no additional implementation is required in the Popcorn environment. Decoupling these two layers will require consensus among all the participant kernels and more messages to determine the ordering. The client request are packed with parameters to execute wait or wake operations.
The entire client operations or the normal Futex operations will be offloaded to the server. This is feasible as we have the user space coherence taken care by previous work [5] and the logic reads the user space address in kernel space. The messaging thread queues the request in the request queue as seen in Figure 5.2. If it is the first time a request is placed for this user address, a dedicated global request handler thread is allocated from the worker thread pool and its identifier is stored in worker_id for servicing further requests. Then, it services the items in the request queue and a server function performs the corresponding operation with the help of wait_attr or wake_attr based on the request type.

5.3.2 Granting Entry into Critical Section

The following explains the different operations and the control flow for granting access to the critical section.

5.3.2.1 Server

The server function caters all the requests and grants access to the critical section. For the wait requests, the futex queue in the server kernel acts as the global queue and reflects the state of all the kernels. If the wait request is from the local client, it queues normally. If it is from a remote client, it populates remote pid and will not have a task_struct. For the wake requests, it queries the global queue to wake the processes. If it is a local process, it queries the local request queue to check if there are any client threads with prior wait request for which the server has responded and the state is between DONE and SLEEP. If so, it sets the clients wake_state to make the wait operation invalid and avoid the race condition. Otherwise, it wakes up the process. If it is a remote process, it issues a remote wake message. The entire operation will be done with the server thread acting as a surrogate for the thread group leader, using the mm_struct of it. This is done to handle page faults that may occur during the processing. Once it is done, it will send a response to the client with a return value.

5.3.2.2 Futex Wait

Figure 5.3 explains the control flow of the wait operation. After getting the response from the server, a client checks if the response contains an error. If so, it will return to the user space. If it is a successful operation and if the client resides in the same kernel as the server it needs to be put to sleep. But, if the client resides in the remote kernel, then it has to be queued in the local futex queue and then put to sleep. Before queuing and scheduling the thread, the variable wake_state is checked to see whether any wake request has been issued by the server between the time client state is DONE and client sets its state to SLEEP. If it was set, then the wait request is invalid and return to user space.
5.3.2.3 Futex Wake

Figure 5.4 explains the control flow of the wake operation. After getting the response from the server, check if there is any error returned from server processing. If the to be resumed process exists in the same kernel as the server, then the server would have woken up the process. So we can safely return to user space. If it exists in the remote kernel, the remote wake will be triggered by the server. Once it reaches the remote, it will check if the value is queued in the local futex queue. If found it will wake up the process and mark the *wake_state*. If not found, then it will check for the state. If the state is not SLEEP, then the client thread has not resumed or scheduled by the local scheduler to reach this state and the remote wake races in front of it. In this scenario, mark the *wake_state* and kick up the process forcefully. It will make the client thread wake up and it will check for the *wake_state* variable for the request to be dropped.
Figure 5.4: Distributed Wake Control Flow

5.3.2.4 Futex Requeue

Figure 5.5 explains the control flow of the requeue operation and it is similar to the wake operation with a minor difference in server side execution. In the server side, it has to check if the remote process has been requeued. If so then the original addr1 will be sent by the server during the remote wake for the remote to check either of addr1 or addr2 to wake up the process.

5.3.3 Process Exit

In the event of the process or thread group crash or exit, robust Futex list is maintained per-thread in user space (maintained by glibc). At do_exit the kernel checks this list to free any lock ups and wake up if any waiters are present. In Popcorn, the main thread is notified about the crash and forces it to go through group exit. This will perform cleanup
activities mentioned before.

During thread group exit, the thread goes through do_exit. This is the place where it performs clean up activities. This includes the following activities:

1. When the thread group exists,
   
   (a) It checks for any running remote thread. If found they are issued signal to gracefully exit the remote as well as the local kernel.

   (b) Then it frees the tasks assigned to Global Worker thread, before releasing it to the pool.

2. When the non-thread group threads who have been migrated at some point exit, they
also check for an entry in Futex hash bucket for any pending remote Futex operations. If found it will be removed from the bucket.

5.3.4 Real-time (Priority Inheritance) Support

Futex supports real-time applications using Priority Inheritance support in kernel space. Currently, Popcorn infrastructure doesn’t have global time stamp across all the kernels. This inhibits the implementation of real-time support. A solution to implement logical global time in a multi-threaded is to have a global time stamp using precision time protocol or real-time clock.

5.3.5 Parallel Messaging Layer

Since the Futex is tightly coupled with the communication layer, the parallel messaging layer will warrant a slight modification in the server side processing. There will be more than one message handler thread receiving requests from the clients. Taking atomic timestamp to order the thread handler would be a way to counter this.

5.3.6 Issues in Scheduling

Currently, the thread migration is done by the same thread calling sched_set_affinity syscall and is confined to the local scheduler to take care of the migration. Since the lack of global scheduling in Popcorn infrastructure, the current implementation does not allow dynamic or scheduler driven scheduling of threads when they are in the middle of synchronization calls. So the thread can either be migrated or allowed to execute further with the futex syscall. To implement it, both the local and remote thread maintains the state as to whether any of the thread is involved in the global synchronization procedure. The state variables futex_state and migration_state are stored in the task_struct. These variables can be modified by taking the lock migration_lock which is also a part of task_struct of the thread. Whenever a syscall is made to Futex subsystem, it sets futex_state variable. When some other thread tries to schedule on behalf of a thread, it checks on the state before proceeding ahead with migration. If it is set, it will cancel the migration and the thread stays with the execution of syscall. If the migration happened before the syscall, the flow will be directed to sleep until the thread returns from the remote kernel.
5.3.7 Process Synchronization

Futex is also used for inter-process synchronization using shared memory area. The key difference is how the application uses a unique identifier for synchronization. For this case in Linux, the unique identifier will be the reference to the struct page pointer and offset within that page of the shared memory mapped to a physical address. The distributed Futex algorithm can be extended in the above-mentioned scenario as well by having to implement memory coherence by distributed shared memory for the file or shared memory objects. Moreover, the reference count of this page is incremented so that it is not swapped out.

5.3.8 Issue of False Sharing

In heterogeneous architecture, the replicated pages are kept coherent by the consistency protocol [26]. If the Futex variables share a page with some other section of a program, the access pattern of the other section could hamper the performance of synchronization. So to avoid this false sharing, we decided to change the glibc library and padded bytes to the common data structure for mutex, condition variables and read-write locks to be of the size of a page (4096 bytes).

5.4 Evaluation

In this section, the performance overhead introduced by the distributed synchronization algorithm is analyzed and quantified for various applications. Being one of the first works to implement distributed SMP synchronization this work solely relies on the comparisons against SMP Linux.

5.4.1 Experimental Setup

To evaluate our mechanism on a homogeneous setup we use a multicore x86 64bit machine with 64 cores, i.e. a 4x AMD Operton 6274 running at 2.4GHz with 128GB of RAM. The evaluation on a heterogeneous setup exploited an x86 64 bit machine with an 8 core, Intel(R) Xeon(R) CPU E5-2695, with 64 GB of RAM and Intel(R) Xeon-Phi 3120A, 57 cores, 4 ways hyper threaded at 1.1GHz, with 6GB of RAM. All the measurements are made with the Time Stamp Counter (TSC) (rdtsc()), corresponds to a single clock cycle, then converted into time in seconds. In order to understand the application characteristics perf tool is used.
The evaluation is categorized into following sections.

1. To verify the functionality with micro-benchmarks and analyze the cost break down.
2. To evaluate macro benchmarks and applications.

### 5.4.2 Microbenchmarks

#### 5.4.2.1 GLIBC Test Suite

The verification of the functionality is done by running the `glibc` test suite (release 2.13) that gcc toolchain provides. The followings are the tests and its description that is used to verify the built mechanism. The other test cases found in their source branch [27] are for inter-process synchronization and is thus excluded.

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tst-mutex2.c</td>
<td>Tests a single <code>pthread_mutex_lock</code> and <code>pthread_mutex_unlock</code> with a barrier inside critical section</td>
</tr>
<tr>
<td>tst-mutex3.c</td>
<td>Tests the above when there is unprecedented signal</td>
</tr>
<tr>
<td>tst-mutex7.c</td>
<td>Tests the above for multiple rounds</td>
</tr>
<tr>
<td>tst-mutex8.c</td>
<td>Tests a single mutex for recursive, error checking and normal mutex attributes</td>
</tr>
<tr>
<td>tst-condition2.c</td>
<td>Tests <code>pthread_cond_wait</code> and <code>pthread_cond_signal</code></td>
</tr>
<tr>
<td>tst-condition3.c</td>
<td>Tests the above when there is unprecedented signal</td>
</tr>
<tr>
<td>tst-condition7.c</td>
<td>Tests the above with thread cancellation</td>
</tr>
<tr>
<td>tst-condition10.c</td>
<td>Tests the above with multiple rounds</td>
</tr>
<tr>
<td>tst-condition14.c</td>
<td>Tests the above with recursive mutex</td>
</tr>
<tr>
<td>tst-condition16.c</td>
<td>Tests for condition wait and broadcast under high contention</td>
</tr>
<tr>
<td>tst-condition18.c</td>
<td>Tests for producer-consumer scenarios</td>
</tr>
<tr>
<td>tst-condition22.c</td>
<td>Tests the above with cancellation and cleanup functions</td>
</tr>
<tr>
<td>tst-barrier3.c</td>
<td>Tests for execution of many threads in using barrier for many rounds.</td>
</tr>
<tr>
<td>tst-barrier4.c</td>
<td>Tests for execution of many threads using multiple barrier for many rounds.</td>
</tr>
</tbody>
</table>

Table 5.2: Test cases executed for evaluating Futex
Measuring Statistics The above-mentioned test cases vary in their synchronization intensiveness. To understand the overhead compared to normal Linux, break down of costs and syscalls on Futex is needed when there are no executing functions or minimal involvement of it. Atomic counters are added to measure the execution time of wait, wake, requeue and server functions in each kernel. Additional counters are added to measure the page faults within server function and the number of retries the user space program has to execute. The types of retries being retried when page fault, when the user address is modified and when the accesses to address returns invalid address. Also, the total messaging cost is measured for evaluating scalability.

Scalability Measurements Lock scalability is measured by the micro-benchmark which does acquire and release lock or spinlock for a loop of 1000000 by increasing the number of threads to measure the worst case scenario. This benchmark is locking intensive as it enters kernel space for arbitration as we increase the number of threads. See Table 5.4 for the perf results of this application. As we increase the threads, there is an increase in access of kernel space data structures like raw_spin_lock for synchronization. Table 5.3 shows the percentage of time spent in Futex in kernel space to categorize this as a high synchronization. Both Linux and Popcorn is better than spinlocks as shown in the Figure 5.6.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Spinlock</th>
<th>Linux</th>
<th>Popcorn</th>
<th>Futex Time</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.10509144</td>
<td>0.165089</td>
<td>0.129121</td>
<td>0.053545186</td>
<td>41.46904234</td>
</tr>
<tr>
<td>4</td>
<td>0.1520194</td>
<td>0.280834</td>
<td>0.095752</td>
<td>0.053807293</td>
<td>56.19431669</td>
</tr>
<tr>
<td>8</td>
<td>0.21927916</td>
<td>0.304467</td>
<td>0.11545</td>
<td>0.051785683</td>
<td>44.85550072</td>
</tr>
<tr>
<td>12</td>
<td>0.28205716</td>
<td>0.29645</td>
<td>0.118607</td>
<td>0.060059277</td>
<td>50.63715979</td>
</tr>
<tr>
<td>16</td>
<td>0.37060881</td>
<td>0.288999</td>
<td>0.184764</td>
<td>0.155711545</td>
<td>84.27578026</td>
</tr>
<tr>
<td>20</td>
<td>0.39470812</td>
<td>0.288999</td>
<td>0.17208</td>
<td>0.134738061</td>
<td>78.2995857</td>
</tr>
<tr>
<td>24</td>
<td>0.4273284</td>
<td>0.284718</td>
<td>0.202157</td>
<td>0.159301931</td>
<td>78.80126894</td>
</tr>
<tr>
<td>28</td>
<td>0.4505775</td>
<td>0.270279</td>
<td>0.212919</td>
<td>0.103474918</td>
<td>48.59814951</td>
</tr>
<tr>
<td>32</td>
<td>0.50947128</td>
<td>0.268902</td>
<td>0.217098</td>
<td>0.084525526</td>
<td>38.93428807</td>
</tr>
</tbody>
</table>

Table 5.3: Lock Scalability

<table>
<thead>
<tr>
<th>Function/Threads</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pthread Userspace</td>
<td>49.65</td>
<td>9.83</td>
<td>10.82</td>
<td>7.31</td>
<td>7.04</td>
<td>6.89</td>
<td>5.33</td>
<td>4.97</td>
<td>4.08</td>
</tr>
<tr>
<td>Raw Spinlock</td>
<td>10.54</td>
<td>33.86</td>
<td>58.47</td>
<td>70.82</td>
<td>75.11</td>
<td>79.39</td>
<td>82</td>
<td>81</td>
<td>79.08</td>
</tr>
<tr>
<td>Others</td>
<td>39.81</td>
<td>56.31</td>
<td>30.71</td>
<td>21.87</td>
<td>17.85</td>
<td>13.72</td>
<td>12.67</td>
<td>14.03</td>
<td>16.84</td>
</tr>
</tbody>
</table>

Table 5.4: Perf results for scalability
Another micro-benchmark that stress on contention is the `tst-barrier3.c` where 100 percent of execution time is spent only on Futex syscalls, where all the thread competes for shared variable, increments it and asked to serialize on a barrier again and again for many rounds. This program in normal Linux spends most of the time in Futex syscall waiting for all the other threads to finish and `perf` results shows less than 20 percent is spent in kernel spinlocks. The average slowdown is around 14.29 percentages as you can see in Table 5.5.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Linux</th>
<th>Syscall</th>
<th>Retry</th>
<th>Popcorn</th>
<th>Messages</th>
<th>Syscall</th>
<th>Retry</th>
<th>Slow Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.003731052</td>
<td>3</td>
<td>1</td>
<td>0.049638918</td>
<td>1610</td>
<td>3556</td>
<td>496</td>
<td>13.30426966</td>
</tr>
<tr>
<td>4</td>
<td>0.010673967</td>
<td>3</td>
<td>1</td>
<td>0.108686277</td>
<td>4747</td>
<td>11038</td>
<td>1814</td>
<td>10.18236952</td>
</tr>
<tr>
<td>8</td>
<td>0.032425481</td>
<td>3</td>
<td>1</td>
<td>0.275907695</td>
<td>16136</td>
<td>37789</td>
<td>6511</td>
<td>8.50897752</td>
</tr>
<tr>
<td>12</td>
<td>0.063707161</td>
<td>3</td>
<td>1</td>
<td>0.510577165</td>
<td>33996</td>
<td>80145</td>
<td>13413</td>
<td>8.014439115</td>
</tr>
<tr>
<td>16</td>
<td>0.109968956</td>
<td>3</td>
<td>1</td>
<td>0.809866728</td>
<td>58537</td>
<td>138979</td>
<td>22102</td>
<td>7.364503179</td>
</tr>
<tr>
<td>20</td>
<td>0.170064448</td>
<td>3</td>
<td>1</td>
<td>1.157080392</td>
<td>89086</td>
<td>211995</td>
<td>33274</td>
<td>6.803775914</td>
</tr>
<tr>
<td>24</td>
<td>0.250649879</td>
<td>3</td>
<td>1</td>
<td>1.569899404</td>
<td>127091</td>
<td>302285</td>
<td>47177</td>
<td>6.263316022</td>
</tr>
<tr>
<td>28</td>
<td>0.32612</td>
<td>3</td>
<td>1</td>
<td>2.04921529</td>
<td>409279</td>
<td>171618</td>
<td>62256</td>
<td>6.283623482</td>
</tr>
<tr>
<td>32</td>
<td>0.428211</td>
<td>3</td>
<td>1</td>
<td>2.5936903</td>
<td>532762</td>
<td>222909</td>
<td>79435</td>
<td>6.057038002</td>
</tr>
</tbody>
</table>
To stress on the number of times the critical section has been accessed under varied contention level, we use the classic producer-consumer benchmark test, `tst-cond18.c`. The test results are taken with one producer and many consumer cores split across different kernels competing for shared resource for a period of 20 seconds. Table 5.6 suggest that Popcorn follows Linux in terms of reduced number of access into the critical section as we increase the threads.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Linux</th>
<th>Popcorn</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>20719</td>
<td>2645</td>
</tr>
<tr>
<td>4</td>
<td>6478</td>
<td>4092</td>
</tr>
<tr>
<td>8</td>
<td>2568</td>
<td>1681</td>
</tr>
<tr>
<td>12</td>
<td>1321</td>
<td>821</td>
</tr>
<tr>
<td>16</td>
<td>811</td>
<td>361</td>
</tr>
<tr>
<td>20</td>
<td>573</td>
<td>152</td>
</tr>
<tr>
<td>24</td>
<td>476</td>
<td>112</td>
</tr>
<tr>
<td>28</td>
<td>421</td>
<td>83</td>
</tr>
<tr>
<td>32</td>
<td>398</td>
<td>67</td>
</tr>
</tbody>
</table>

Table 5.6: Producer-Consumer: Number of access into critical section

<table>
<thead>
<tr>
<th>Linux Syscalls</th>
<th>Retry</th>
<th>Popcorn Syscalls</th>
<th>Retry</th>
<th>Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.67</td>
<td>1.33</td>
<td>1548249.33</td>
<td>632723.11</td>
<td>3468130.89</td>
</tr>
</tbody>
</table>

Table 5.7: Average syscalls and retries for producer-consumer

**Low Contention** We test the implementation with `tst-mutex2.c` which exhibits low contention. It has two threads which are serialized by a barrier after entering the critical section. Table 5.8 suggests that only 4 percentage time is spent on Futex for this application. When we measured the popcorn performance the additional cost introduced by the Futex layer is due to the server execution function and messaging cost incurred by every client before it can be queued locally. On an average, there were 13 additional messages. Response time is calculated by the sum of server execution time plus the total message cost. On an average 4.0641 percentage of total time is spent on this part as seen in Table 5.9. There is also an increase in the number of syscall and retries as mentioned in Table 5.11. All the below-mentioned retries indicate the user space variable has been modified and the execution has to be repeated. The average slowdown of the application including migration cost, messaging cost and synchronization cost is 7.43 times.
The same set of evaluation is done on heterogeneous systems. On an average, there were 12.8 additional messages as mentioned in Table 5.11. Response time is calculated by the sum of server execution time plus the total message cost. On an average 14.8641 percentage of total time is spent on this part. See Table 5.10. There is also an increase in the number of syscall and retry as seen in homogeneous. The average slowdown of the application including migration cost, messaging cost and synchronization cost is 931 times. Interesting to note that the breakdown cost for Futex is lower when there is no page fault occurs. The server execution function has to fix and retry the operation if there is a page fault and the cost for resolving it would be added to the execution function. See Figure 5.8.

**Results**  
We can infer from the high contention experiments that if we increase the number of threads the Futex syscalls increase exponentially as compared to 3 syscalls in normal SMP Linux. It can be inferred that when the application resolves the contention in user space Linux performs better. But if it has to come to kernel space for resolving contention more often, shared data structures or locks become a bottleneck. This is where Popcorns multikernel mechanism will alleviate contention. The number of access to the critical section under high contention tapers down as the number of threads increase for both Linux and
% Function
83.16 kmem_cache_free
7.82 update_rq_clock
4.12 futex_wait
4.9 others

Table 5.8: Perf results for mutex between two threads

<table>
<thead>
<tr>
<th>Linux</th>
<th>Popcorn</th>
<th>Server Time</th>
<th>Total Response</th>
<th>Total Messages</th>
<th>Page Fault</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000344759</td>
<td>0.003821698</td>
<td>0.000139628</td>
<td>0.000209373</td>
<td>12</td>
<td>0</td>
<td>5.47853</td>
</tr>
<tr>
<td>0.000434625</td>
<td>0.00184392</td>
<td>2.07367E-05</td>
<td>9.04817E-05</td>
<td>12</td>
<td>0</td>
<td>4.907027</td>
</tr>
<tr>
<td>0.000374691</td>
<td>0.002566341</td>
<td>3.43238E-05</td>
<td>0.000115693</td>
<td>14</td>
<td>0</td>
<td>4.50808</td>
</tr>
<tr>
<td>0.000399792</td>
<td>0.003416201</td>
<td>3.41963E-05</td>
<td>0.000115565</td>
<td>14</td>
<td>0</td>
<td>3.38286</td>
</tr>
<tr>
<td>0.000362608</td>
<td>0.002435951</td>
<td>2.06133E-05</td>
<td>9.03583E-05</td>
<td>12</td>
<td>0</td>
<td>3.70936</td>
</tr>
<tr>
<td>0.000390968</td>
<td>0.002478114</td>
<td>2.61385E-05</td>
<td>9.58588E-05</td>
<td>12</td>
<td>0</td>
<td>3.86821</td>
</tr>
<tr>
<td>0.000381179</td>
<td>0.003069868</td>
<td>2.97583E-05</td>
<td>0.00011128</td>
<td>14</td>
<td>0</td>
<td>3.61994</td>
</tr>
<tr>
<td>0.000331475</td>
<td>0.003075079</td>
<td>3.37813E-05</td>
<td>0.00011515</td>
<td>14</td>
<td>0</td>
<td>3.74463</td>
</tr>
<tr>
<td>0.000382518</td>
<td>0.002429496</td>
<td>2.09425E-05</td>
<td>9.06875E-05</td>
<td>12</td>
<td>0</td>
<td>3.73276</td>
</tr>
<tr>
<td>0.000360875</td>
<td>0.002477886</td>
<td>2.16896E-05</td>
<td>9.14346E-05</td>
<td>12</td>
<td>0</td>
<td>3.69002</td>
</tr>
</tbody>
</table>

Table 5.9: Mutex between two threads serialized by barriers

<table>
<thead>
<tr>
<th>Linux</th>
<th>Popcorn</th>
<th>Server Time</th>
<th>Messages</th>
<th>Page Fault</th>
<th>Total Time</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000141084</td>
<td>0.31936329</td>
<td>0.094137754</td>
<td>14</td>
<td>1</td>
<td>0.0941481</td>
<td>29.4793813</td>
</tr>
<tr>
<td>0.000146614</td>
<td>0.218115195</td>
<td>0.092423323</td>
<td>14</td>
<td>1</td>
<td>0.09243369</td>
<td>42.37837215</td>
</tr>
<tr>
<td>0.000111242</td>
<td>0.032259099</td>
<td>0.000779035</td>
<td>14</td>
<td>1</td>
<td>0.000789382</td>
<td>2.447004382</td>
</tr>
<tr>
<td>0.000189483</td>
<td>0.1300388</td>
<td>0.093297935</td>
<td>14</td>
<td>1</td>
<td>0.093308282</td>
<td>71.75418534</td>
</tr>
<tr>
<td>0.000164223</td>
<td>0.125540928</td>
<td>0.000702936</td>
<td>14</td>
<td>1</td>
<td>0.000713283</td>
<td>0.568167548</td>
</tr>
<tr>
<td>0.000111962</td>
<td>0.036472599</td>
<td>0.00013455</td>
<td>10</td>
<td>0</td>
<td>2.08545E-05</td>
<td>0.057153591</td>
</tr>
<tr>
<td>0.00014063</td>
<td>0.22725088</td>
<td>0.00012565</td>
<td>10</td>
<td>0</td>
<td>1.99554E-05</td>
<td>0.00878122</td>
</tr>
<tr>
<td>0.00015343</td>
<td>0.0248347</td>
<td>0.00006285</td>
<td>8</td>
<td>0</td>
<td>1.21973E-05</td>
<td>0.049114022</td>
</tr>
<tr>
<td>0.000159013</td>
<td>0.022698083</td>
<td>8.3275E-06</td>
<td>10</td>
<td>0</td>
<td>1.57179E-05</td>
<td>0.0692477</td>
</tr>
<tr>
<td>0.000141993</td>
<td>0.13200154</td>
<td>1.47888E-05</td>
<td>10</td>
<td>0</td>
<td>2.21792E-05</td>
<td>0.01680219</td>
</tr>
</tbody>
</table>

Table 5.10: Mutex between two threads serialized by barriers in Heterogeneous

<table>
<thead>
<tr>
<th>Platform</th>
<th>Linux Syscalls</th>
<th>Retry</th>
<th>Page Fault</th>
<th>Popcorn Syscalls</th>
<th>Retry</th>
<th>Server Calls</th>
<th>Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heterogenous</td>
<td>5</td>
<td>1</td>
<td>9.4</td>
<td>2</td>
<td>0.7</td>
<td>6.4</td>
<td>12.8</td>
</tr>
<tr>
<td>Homogenous</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>9</td>
<td>2.4</td>
<td>6.4</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 5.11: Average syscalls and retries for Mutex between two threads
5.4.3 Macro Benchmarks

5.4.3.1 Homogeneous

We test the NAS Parallel Benchmark suite [30] which run the parallel application on SMP machines on homogeneous setup. This has proven to give best results with Popcorn architecture. The workload characterization of the following benchmarks can be found in our previous work [5]. In our previous work, these applications were adapted to Popcorn to make use of the custom threading library that is used in the Popcorn user space, cthread. Cthread was developed by Antonio Barbalace to replace the pthread library. We compare the synchronization overheads introduced by this implementation against this baseline by Popcorn. But the number of access is high for fewer threads. This is where the existing Linux has the advantage over all the other mechanisms as everything is resolved in user space.
running Barrelish OpenMP library (bomp) which depends upon the pthread library. Our evaluations are limited to applications of type class A. These require a medium size input data set. The memory footprint of these application increases with other classes, requiring more allocation of memory for each kernel.

**IS-BOMP** IS is a parallel integer sorting benchmark designed to test random memory access. IS-bomp is its OpenMP version which run on top of bomp. Bomp is a minimal OpenMP implementation based on the pthread library. We test it against Linux and is-pomp, a Popcorn adaptation of is-bomp workload designed by NASA in the NAS Parallel Benchmark suite without the Futex synchronization. We use `perf` tool to measure the breakdown in vanilla Linux. We measure the additional messages, syscalls and timing overheads introduced by Futex layer. `Perf` measurement shows only 0.01 percentage of the application time is invested in doing `sys_futex` operation. The syscalls mentioned in the Table 5.12 are due to this call made during the thread exit for unlocking the pages used by the Futex variables. This application can be categorized as less synchronization intensive. It uses spin-lock mechanisms `bomp_lock` and `bomp_barrier_wait` for synchronization as the application’s compute part is offloaded and the main thread waits for all the worker threads to report. The overhead mentioned in Figure 5.9 is due to additional syscalls introduced by using `pthread` library. This can be verified in Table 5.13 as the average time spent by the Futex layer is 0.026 percentage of the entire compute time.

<table>
<thead>
<tr>
<th>Linux Futex</th>
<th>Retry</th>
<th>Popcorn</th>
<th>Retry</th>
<th>Server Call</th>
<th>Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>11</td>
<td>0</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>27</td>
<td>0</td>
<td>13</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>43</td>
<td>0</td>
<td>22</td>
<td>55</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>65</td>
<td>1</td>
<td>35</td>
<td>86</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>79</td>
<td>1</td>
<td>41</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>95</td>
<td>1</td>
<td>49</td>
<td>121</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>116</td>
<td>5</td>
<td>63</td>
<td>154</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>131</td>
<td>1</td>
<td>69</td>
<td>171</td>
</tr>
</tbody>
</table>

Table 5.12: Syscall count comparison for IS-BOMP

**CG-BOMP** CG is a parallel conjugate gradient benchmark designed to test irregular memory access. We test it against cg-pomp, a Popcorn adaptation of the cg-omp workload designed by NASA and normal cg-omp run on Linux. We measure the additional messages, syscalls and timing overheads introduced by Futex layer. The average time spent by the Futex layer is 0.052 percentage of the compute time as mentioned in Table 5.15. This can also be categorized as less synchronization intensive and has similar characteristics as is-bomp. All
Figure 5.9: Performance comparison for IS-BOMP

<table>
<thead>
<tr>
<th>Cores</th>
<th>Linux</th>
<th>Popcorn-Pomp</th>
<th>Popcorn</th>
<th>Server</th>
<th>Messages</th>
<th>Futex Time</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.532725133</td>
<td>1.546854305</td>
<td>1.890884788</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1.185839647</td>
<td>0.990274611</td>
<td>1.537808067</td>
<td>1.90583E-05</td>
<td>8.88033E-05</td>
<td>0.000107862</td>
<td>0.007013988</td>
</tr>
<tr>
<td>8</td>
<td>1.747790154</td>
<td>1.034489404</td>
<td>1.54740447</td>
<td>6.03579E-05</td>
<td>0.000246345</td>
<td>0.000306703</td>
<td>0.019820448</td>
</tr>
<tr>
<td>12</td>
<td>1.740106111</td>
<td>1.269885167</td>
<td>2.179110081</td>
<td>8.05142E-05</td>
<td>0.000400179</td>
<td>0.000480693</td>
<td>0.022132266</td>
</tr>
<tr>
<td>16</td>
<td>2.051327006</td>
<td>1.567500687</td>
<td>2.570329176</td>
<td>0.000128091</td>
<td>0.00062793</td>
<td>0.000756021</td>
<td>0.029413386</td>
</tr>
<tr>
<td>20</td>
<td>2.424451337</td>
<td>1.9100445</td>
<td>3.106192121</td>
<td>0.000150049</td>
<td>0.00073707</td>
<td>0.000887119</td>
<td>0.028559687</td>
</tr>
<tr>
<td>24</td>
<td>2.712741487</td>
<td>2.197539404</td>
<td>3.714065702</td>
<td>0.0002086</td>
<td>0.000911862</td>
<td>0.001120461</td>
<td>0.030168051</td>
</tr>
<tr>
<td>28</td>
<td>2.715581884</td>
<td>2.608048263</td>
<td>3.757911012</td>
<td>0.00023844</td>
<td>0.001133501</td>
<td>0.001371942</td>
<td>0.036508094</td>
</tr>
<tr>
<td>32</td>
<td>2.924940845</td>
<td>2.869483505</td>
<td>3.881062504</td>
<td>0.000293877</td>
<td>0.001287743</td>
<td>0.00158162</td>
<td>0.040752232</td>
</tr>
</tbody>
</table>

Table 5.13: Cost breakdown for IS-BOMP

the syscalls made for Futex are during thread exit when the main thread coordinates with the worker threads. This minimal overhead doesn’t affect the performance as we can see in the Figure 5.10.

**FT-BOMP** FT is a parallel discrete 3D fast Fourier Transform benchmark designed to test all communications between computing nodes. We test it against ft-pomp, a Popcorn adaptation of the ft-omp workload designed by NASA and normal ft-omp run on Linux. We
Figure 5.10: Performance comparison for CG-BOMP

<table>
<thead>
<tr>
<th></th>
<th>Linux Futex</th>
<th>Retry</th>
<th>Popcorn Futex</th>
<th>Retry</th>
<th>Server Call</th>
<th>Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>25</td>
<td>1</td>
<td>10</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>53</td>
<td>2</td>
<td>18</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>77</td>
<td>1</td>
<td>22</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>124</td>
<td>4</td>
<td>37</td>
<td>88</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>139</td>
<td>3</td>
<td>44</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>167</td>
<td>2</td>
<td>52</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>190</td>
<td>2</td>
<td>55</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>211</td>
<td>3</td>
<td>66</td>
<td>162</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.14: Syscall count comparison for CG-BOMP

measure the additional messages, syscalls and timing overheads introduced by Futex layer. Predictable characteristics are seen similar to the previous two applications. The Figure 5.11 shows there is small overhead introduced by additional syscalls.
Figure 5.11: Performance comparison for FT-BOMP

![Performance comparison for FT-BOMP](image)

Table 5.15: Cost breakdown for CG-BOMP

<table>
<thead>
<tr>
<th>Cores</th>
<th>Linux</th>
<th>Popcorn-Pomp</th>
<th>Popcorn</th>
<th>Server</th>
<th>Messages</th>
<th>Futex</th>
<th>Time</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10.21337779</td>
<td>55.17479004</td>
<td>55.24725629</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>10.20921362</td>
<td>42.60026399</td>
<td>43.02592286</td>
<td>8.60488E-05</td>
<td>0.000213915</td>
<td>0.000299963</td>
<td>0.000697169</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>10.11186525</td>
<td>25.5517271</td>
<td>26.75160635</td>
<td>7.93025E-05</td>
<td>0.000329222</td>
<td>0.000408525</td>
<td>0.001527103</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>2.873350273</td>
<td>21.13950376</td>
<td>21.13966332</td>
<td>9.05896E-05</td>
<td>0.000404442</td>
<td>0.000495032</td>
<td>0.00234172</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0.679101381</td>
<td>16.77972644</td>
<td>16.98240446</td>
<td>0.000139651</td>
<td>0.0006551114</td>
<td>0.000790765</td>
<td>0.00465378</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0.579018216</td>
<td>14.41506703</td>
<td>15.03308584</td>
<td>0.00016279</td>
<td>0.000778871</td>
<td>0.000941662</td>
<td>0.006263928</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>0.647140029</td>
<td>13.2340893</td>
<td>13.3379751</td>
<td>0.000247297</td>
<td>0.000991244</td>
<td>0.001238541</td>
<td>0.009285824</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>0.475293745</td>
<td>12.70656919</td>
<td>12.42965933</td>
<td>0.000225487</td>
<td>0.00101593</td>
<td>0.001241418</td>
<td>0.009987542</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>0.495815115</td>
<td>12.0207125</td>
<td>12.32838254</td>
<td>0.0002722</td>
<td>0.001213758</td>
<td>0.001485958</td>
<td>0.012053195</td>
<td></td>
</tr>
</tbody>
</table>

5.4.3.2 Heterogeneous

The following are the benchmarks run on a heterogeneous platform. This is to showcase that the existing prototype works with different applications and a means to analyze the overhead introduced by the Futex layer. These applications are made run without compiler refactoring [6]. Futex usages metric can be used by the partitioner to analyze and make a
better judgment on running or migrating the threads among other kernels. The following benchmarks are chosen for the following reasons:

- Fluidanimate: A fluid particle simulator where threads are involved in moderate contention.
- Pbzip2: A data compression software where different threads perform different tasks in parallel.
- Matrix multiplication: A matrix multiplication benchmark which is a data parallel application with low contention.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Linux</th>
<th>Popcorn Pomp</th>
<th>Popcorn Futex</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>18.36830111</td>
<td>23.60239775</td>
<td>28.01514059</td>
</tr>
<tr>
<td>4</td>
<td>13.24256832</td>
<td>16.72030099</td>
<td>20.99979221</td>
</tr>
<tr>
<td>8</td>
<td>12.2825577</td>
<td>12.27254147</td>
<td>17.99092069</td>
</tr>
<tr>
<td>12</td>
<td>11.52471369</td>
<td>11.6712044</td>
<td>18.0209019</td>
</tr>
<tr>
<td>16</td>
<td>11.05736385</td>
<td>11.26124741</td>
<td>17.30451123</td>
</tr>
<tr>
<td>20</td>
<td>9.909629858</td>
<td>11.36023189</td>
<td>18.26032205</td>
</tr>
<tr>
<td>24</td>
<td>9.190151118</td>
<td>11.27690894</td>
<td>18.46187288</td>
</tr>
<tr>
<td>28</td>
<td>8.637570373</td>
<td>11.3611029</td>
<td>18.75670231</td>
</tr>
<tr>
<td>32</td>
<td>8.594912838</td>
<td>12.18228082</td>
<td>19.04530218</td>
</tr>
</tbody>
</table>

Table 5.16: Cost breakdown for FT-BOMP

5.4.3.3 Fluidanimate

Fluidanimate is an application from PARSEC [31]. It simulates an incompressible fluid for interactive animation purposes with fine-grained parallelism. So it stresses on synchronization and communication. See Table 2.2 in the above-cited the paper. We use the small set of data, around 5000 particles, with one frame to animate for our evaluation. The application requires the threads be allocated in power of two. We allocated one thread in Xeon and
the rest in Xeon-Phi with the maximum threads being 32. Perf results show that, as the threads are increased it increases synchronization calls. Most of them are resolved in user space and under maximum contention it invests 20 percentage of time in kernel space. As we have seen with micro-benchmarks, whenever contention is resolved in user space, Linux performs far better.

### 5.4.3.4 PBZIP2

PBZIP2 is a parallel implementation of the bzip2 block-sorting file compressor. This application uses pthread threading constructs to achieve performance benefit on SMP machines. This has two I/O threads doing the file read and write operation and and multiple compute threads doing the actual compression. We decided to compress 1 Gb file size by keeping I/O threads in Xeon and migrate compute threads to Xeon-Phi. We did a test with reading the entire file and split it between threads. This will reduce the synchronization between I/O threads and compression threads. If the file is split dynamically, synchronization will increase as well as the data transfer has to happen frequently. We tested the application with 2 compression threads one in Xeon and another in Xeon Phi. As seen in the Tables 5.19 and 5.21 the average Futex overhead is lesser, 1.08% compared with 14.73%, when the entire file is read before compression then reading dynamically as it would introduce synchronization communication between kernels. Since there are more syscalls Table 5.20, around 4 times more, applications run in kernel space more than when running in Linux.

![Table 5.18: Fluidanimate](image)

![Table 5.19: Compression starts after entire file is read](image)
Table 5.20: Average syscalls and retries for PBZIP2

<table>
<thead>
<tr>
<th>Mode</th>
<th>Linux Syscalls</th>
<th>Retry</th>
<th>Popcorn Syscalls</th>
<th>Retry</th>
<th>Server calls</th>
<th>Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Entirely</td>
<td>22</td>
<td>1.2</td>
<td>82.6</td>
<td>11.6</td>
<td>57</td>
<td>129.6</td>
</tr>
<tr>
<td>Read Dynamically</td>
<td>1379</td>
<td>4</td>
<td>4873.2</td>
<td>842.4</td>
<td>4682.4</td>
<td>10419.6</td>
</tr>
</tbody>
</table>

Table 5.21: Compression starts when the file is dynamically read

5.4.3.5 Matrix Multiplication

Matrix multiplication is a map reduce application from Metis [32]. This workload allocates large amounts of memory to hold temporary tables, stressing the kernel memory data structures. This also uses data parallel parallelization and less of synchronization. We tested with the multiplication of 1024 rows and columns with 4 threads on Xeon and 228 threads on Xeon-Phi. Results suggest that Popcorn is 5.78 times slower than Linux. As this application is less communication intensive, only 0.0053 percentage of computing is invested on Futex and 5812 additional messages are invoked due to synchronization. As reported by other benchmarks there is a proportionality between the overhead added by the futex mechanism and the amount of synchronization required by the application.

Table 5.22: Matrix multiplication
Chapter 6

Process Management

6.1 Introduction

This chapter introduces the management features developed for the user space program to see the resources available across all the kernels. This includes features like kernel discovery, unique process identification space, process virtual file system, and namespaces for migration which enables the benchmarks and application written to use these for its execution.

6.2 Design Motivation

Popcorn provides the illusion that all processes running amongst the different kernels are running on the same operating environment. The goal is to provide the user with the process management tools (e.g. ps, kill on Unix-like systems) operate on all processes across the kernels. The following utilities are implemented:

1. Modifying the existing /proc file system to display the details of the process, memory, and CPU of the remote kernel for the user application to look for the entire system details for optimized execution.

2. In order to establish communication through signals or synchronization, it will act as a lookup dictionary.
6.3 Implementation

6.3.1 Kernel Discovery

As with the Popcorn architecture, a primary kernel will be booted initially. During the boot process, it will broadcast messages with the current CPUs detail to all the CPUs on the platform to check their availability. When the secondary kernel starts to boot up, similar broadcast messages are sent. If there are kernels alive, they process this discovery messages and register the discoverer CPU information in a local dictionary and then sends to the requester kernel to record its details. When a kernel dies, it again sends a broadcast message to wipe out all the details in their local dictionary. Currently, architecture related data like allocated page frame number range, CPU specifics, and network interface details are shared among the kernel by this module.

6.3.2 Single IP Address

Distributed single IP address is essential for the migrated thread to share the network interface with the original kernel and continue accessing the same network interface.

This is done by replicating the device driver set up for the supported network interface. This proxy module has to be written specifically for the network device. We implemented the support for the Intel’s 82576 GbE board. When the kernel boots up, the kernel discovery module starts these proxy modules that are known to the local kernel. Each proxy module inquiries the devices associated with the local generic socket namespace init.net and obtains device and IP address details. It then broadcasts to all the other remote kernels with the network device details and they do same. When a proxy module receives back any device information from any other kernel, it does the following 3 steps to replicate the original kernel set up:

1. Register a dummy device through pci_register_driver. This is done once for a particular device type and for a given remote kernel. These proxy devices will act as a slave to the original device driver located in the owner kernel and cannot control the device. Then add this dummy device as a network device, copy original device details, initialize it as inet device and set the status as the physical link is present.

2. Replicate the inet address on to the dummy device. This is done by inet_set_ifa with the help of remote IP, mask, and broadcast address.

3. Set the routing table. This is done by calling ip_rt_ioctl with the remote gateway, mask and its destination.
The above steps are necessary to create an entry in kernel for `procnetdev` and `procnetroute` query the proxy device name and its routing table. This enables any local thread or migrated thread to access and share the network interface from an application standpoint. For marshaling incoming packets, the server kernel will have to employ interrupt routing to a given CPU/kernel.

### 6.3.2.1 Socket State Replication

This module uses the existing infrastructure built for thread migration. All sockets are associated with the process as files. So it will be attached to the file descriptor table. The server thread can be migrated just after it has been allocated a file descriptor, or bound to an address, or still listening to the connection or in the process of accepting live connections; the last one will be termed as live migration.

### 6.3.2.2 Socket File Descriptor Migration

The file descriptors associated with create and accept process needs to be migrated. The original file descriptors, with which the application will identify a particular socket will be installed in `task_struct` as surrogate file descriptors. Based on one the above-mentioned states, the migrated thread will establish the same state as in the original. For the creation of a socket, the new file descriptors are installed, mapped along with surrogate descriptors and shall never be available to the user. We modify the file descriptor lookup, so when the application syscalls with old file descriptor the new file descriptor return to the kernel functions.

### 6.3.3 Unique Process ID

To identify remote processes, the Process ID are not enumerated as in Linux but are given globally unique IDs that make them identifiable across all kernel instances. In Linux, PIDs are generated by an integer ID manager [33] for device names; IPC names and others. They are a 32-bit integer. On top of this we create the global PID by augmenting the bits to Kernel Id and the actual PID.

The kernel ID in homogeneous systems currently assigned from the value `present_mask` parameter which is given at the boot arguments. When a thread is created it gets a PID, this kernel ID gets masked in the upper 8 bits and remaining 24 bits are used for actual PID. This static partitioning can support 256 cores and $2^{24}$ PID’s in a kernel. To address the core count explosion in future, we can change the 32-bit integer to 64-bit integer to accommodate more cores. Anytime a thread is migrated on a different kernel it gets a new PID and the same process applies to them.
In heterogeneous systems, additional bits for an ISA or CPU group is reserved. So the global PID will be a result of masking ISA group + Kernel ID in the group + local PID. These can also be passed from boot arguments.

When the threads are migrated they should not appear to have left the process space. The migrated thread will be assigned a new local PID, but its task_struct will hold the origin PID which is the PID in the first kernel and this kernel shadow process (master kernel for this PID) gets updated with the latest remote kernel where the thread is running. Any PID related syscalls, will resolve to the origin PID and if the task is not running in the current kernel, it will be directed to the master kernel which will in turn redirect to the latest running kernel.

This will nullify the overhead of allocating and de-allocating process space between the kernels; which would incur messages and keeps the process space functionality local. The Popcorn architecture resembles like a cluster with all its the subsystems partitioned across multiple kernels. In this scenario, user applications in any kernel should have a global view/monitor and access to all the resources as it is originally a single machine and will be better for scheduling reasons. This is the reason why we didn’t go for design principles like BProc or distributed hash table mechanism like chord [34]. Popcorn provides inherent isolation. Similar to process space we can make other resources global when isolation is no longer needed.

### 6.3.4 Proc File System

This virtual file system is the control and information center for the kernel. The proc file system represents a file access method to the components that exists in Linux. Their data is generated dynamically by querying the OS state and are displayed with the help of seq_files. These are essential files from where commands like ps, top fetches its information to display or access global data.

Currently, we have modified the proc file system to display the list of remote PIDs and the read-only files associated with each task like stat, cpuset, comm, cmdline and others. No data will be populated initially. When a user accesses these files, then the associated data is fetched from the remote kernels, which is identified by kernel discovery phase, on demand. Architecture dependent information like cpuinfo and meminfo has generic data structures to query the data and display. For clustering environments, the query is sent only to the master CPU to fetch as master CPU within a cluster will have all the relevant data of the CPUs within the cluster.

All these data will have to be accessed via the messaging layer. The cost of accessing
<table>
<thead>
<tr>
<th>Kernel</th>
<th>response Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.036</td>
</tr>
<tr>
<td>2</td>
<td>0.043</td>
</tr>
<tr>
<td>3</td>
<td>0.064</td>
</tr>
<tr>
<td>4</td>
<td>0.112</td>
</tr>
<tr>
<td>8</td>
<td>0.251</td>
</tr>
</tbody>
</table>

Table 6.1: Average Response Time for `ps` command

CPU info and other architectural data are local as when each kernel is booted, the kernel has a copy of their information. The response time for `ps` command will have the following delay detailed in the table 6.1.

### 6.3.5 Namespaces

Namespaces were introduced into SMP Linux to create sub-environments, as a lightweight virtualization alternative implemented at the OS level. Kerrighed [35] uses namespaces (containers) to migrate applications in a cluster by reproducing the same contained environment on different kernels. Popcorn uses namespaces to provide a single environment shared between kernels. Each kernels namespace is kept consistent with the others through the messaging layer.

After the kernels are connected via messaging layer, static Popcorn namespaces are created in each kernel. A global `nsproxy` structure is then made to point to Popcorns utfs, mount, IPC, PID, network and CPU namespace objects. Namespaces on each kernel are updated whenever a new kernel joins the replicated-kernel OS. Since it is updated whenever a kernel starts and exits, the overhead is minimal and the lookup is similar to Linux.

Because Popcorn namespaces get created by an asynchronous event rather than the creation of a task, instead of using the common `/proc/PID/ns` interface we added `/proc/popcorn`. Tasks join Popcorn by associating to its namespaces through the use of the `setns` syscall, which has been updated to work with statically created namespaces. When a task is migrated to another kernel it starts executing only after being (automatically) associated to Popcorn namespaces.
Chapter 7

Conclusion

One of the main goals of this Thesis is to provide the same level of programmability available on a SMP OS on a replicated-kernel OS, through a familiar POSIX interface. In order to achieve it, the existing user space shared memory and communication layer needs to be stitched with thread communication features and process management features to provide the single system image for the application. Prior research in single system image has opted for dedicated servers to host these services, or have deviated from SMP programming model forcing the user to rewrite applications or restricted to process synchronization and communication.

In this thesis, Popcorn Linux was extended to provide thread communication and process management services by enabling these services in all the kernels to support SMP programmability. The signals distributed service lets the threads distributed across kernel communicate as in normal SMP program. The distributed synchronization layer lets the threads distributed across kernels maintain consistency. The namespaces and process management services let the application know about the available resources across other kernels for migration. These mechanisms have been successfully verified on a homogeneous x86 setup and a heterogeneous-ISA platform. These mechanisms were validated by modeling and exhaustive testing. Though the performance was not the driving goal of the project, we evaluated the synchronization overheads for different contention loads, scalability and testing different real world applications.

Popcorn can run different applications, as discussed in section 5.4. The performance numbers are taken to understand the characteristics of the different application and the overhead introduced by the layer. The average number of messages used to resolve a single synchronization syscall is around 1.5 to 2.5 messages. But the reason for the increase in number of issued syscall is due to the retries issued by the server while reading the user space value, which may have been modified; causing the operation to be invalid and the client has to retry the operation again. In a distributed setup, there will be some delay for write intensive operations as the layer sits on top of distributed cache coherency layer and messaging layer. In
the heterogeneous environment, the server execution function incurs additional overhead for resolving page faults using the replicated cache coherency layer. Our scalability results show that for high contention applications, Popcorn can remove bottlenecks caused by shared data structure and locks. But most applications try to minimize the synchronization communication and resolve it in user space. This needs to be studied further to conclude whether a synchronization primitive based on cache coherency layer is the option for performance in multiple kernel architectures.

7.1 Future Work

7.1.1 Shared Memory Synchronization Variants

Further research have been made to understand whether the synchronization model based on SMP is the way to go for distributed heterogeneous systems. It would be interesting to evaluate existing implementation on a shared memory with relaxed consistency. Also, following the integration trend of CPU and GPUs in which they can both share a common memory, we foresee that OS-capable CPU will maybe share the same memory therefore removing part of the overhead that we are currently paying for the synchronization therefore making the system faster.

7.1.2 Alternative Approach to Synchronization

Profiling the real world applications for threading patterns like pipelining, and helper models should be done. This will help to identify a different pattern in which locking is employed. This can help us redesign the algorithm with some policy or domain based synchronization and let the application characterization decide the policy. Another parallel methodology would be to move away from SMP based synchronization and decouple with the underlying layer at the cost of more messages or the layer can be brought inside kernel space incurring more context switching. We need to apply the existing well known distributed algorithms. This should be done to evaluate the best fit and determine the trade off index between performance and programmability.

7.1.3 Process Management

For implementing synchronization at the kernel level subsystems or process synchronization, distributed shared memory resource management should be incorporated in this layer. This will enable us to execute process communication through pipes and shared memory. Performance metrics like usage of messages, contention between threads and memory subsystem
parameters should be queried and stored for the implementation of global scheduling so that Popcorn can make well-formed decisions. The existing socket state layer can be exploited to remap the interrupt across kernels. This will enable us to execute network driven applications across kernels.
Bibliography


[4] SSRG, “”thread migration in a replicated-kernel os” david katz, antonio barbalace, saif ansary, akshay ravichandran and binoy ravindran, under submission icdcs2015.”


